Course Introduction

Introduction to Programming

7.9.2023

Outline

- 5 ECTs.
- 7.9. - 19.10.2022
- One "lecture" per week at Thursdays, 10.15
- Additional workshop times, to be listed in MOOC.fi

- Focus on independent work

Teachers

Erkki, Timo

Eero, Erika, Henna, Isabel, Lotta-Linnea, Sohvi

Junior instructors

- erkki.kaila@helsinki.fi - use email primarily
- ohjelmoinnin-mooc@helsinki.fi - concerning material (GIT issues or PR's work also nicely!)
- C213, Exactum (though I'm rarely there)
- +358 50 3454 325 (if I remember to switch it on occasionally)

What are we learning?

Basic programming concepts using **Python** as a programming language

Continued in Advanced Programming Course in the second period

Contents

Week	Topic
1	Introduction, I/O, variables, conditional statements
2	More conditions, loops
3	More loops, strings
4	Own functions
5	Lists and other data structures
6	Files
7	External libraries

Initial Exam

If you know programming already and haven't completed similar course with MOOC.fi or in any other Finnish higher institute, you can pass the course with initial exam

Exam takes place at Friday 8.9. at 12.00 online

Register via email **today**: erkki.kaila@helsinki.fi

What is programming?

Writing **programs** using a **programming language**

Why?

Although there is a lot of discussion about artificial intelligence and machine learning etc., computers are still quite "stupid"

However, they are great at executing simple tasks repeatedly with great speed

With programming, these kind of tasks can be used to construct solutions to complex problems

Real Life Example

Learning analytics data from elementary school level

Calculations and students' answers to calculations

Goal was to filter the needed rows from the entire data

Example (cont.)

Several tasks, such as

"Find all subtractions where the first number is less than 21 and the second one has only one digit"

...so these are ok:

11 - 6
19 - 4

...but these need to be filtered out:

24 - 7
19 - 11

Short excerpt from the file

```
SUBTRACTION,70-30,1,48553,anon9159,"2016-01-18 10:29:11"
SUBTRACTION,70-60,1,16787,anon9159,"2016-01-18 10:29:11"
SUBTRACTION,90-40,1,12990,anon9159,"2016-01-18 10:29:11"
SUBTRACTION,14-6,1,35139,anon9159,"2016-01-18 10:29:11"
SUBTRACTION,50-30,1,6575,anon9159,"2016-01-18 10:29:11"
SUBTRACTION,19-3,1,23239,anon9159,"2016-01-18 10:29:11"
SUBTRACTION,80-60,1,7785,anon9159,"2016-01-18 10:29:11"
```

Approx. 250 000+ lines per file, several files

Program

Python program which filtered 8 different kind of tasks from all files

Writing and testing the program took around 30 minutes

Handling 5 million rows of data took less than a minute

What is Python?

Translated programming language

Used for various purposes, very popular in e.g. AI and bio technology

Powerful and versatile, still simple syntax

What Do We Do in Course?

Two components:

1. Thursday shared sessions (such as this)
2. Completing exercises in MOOC.fi

Finally, an exam

Workload

Learning to program requires a lot of work

Workload is around 10 to 20 h per week

...although with previous experience the first weeks may be completed faster

Course Material

All course material is found here

https://programming-23.mooc.fi/

Workshops

Exercises are completed independently, but you don't need to complete them alone

Help can be asked in workshops and in Discord, detailed information is found in MOOC.fi

Is Programming Difficult?

It may appear to be at first.

Different to other skills

- Not math, although math is important in some areas of programming
- Not pure logic either

All research emphasizes the importance of active learning

TMC

Programming tasks, quizzes

Automatic assessment, immediate feedback

All exercises can be retaken as many times as needed

First weeks in browser, after that with Visual Studio Code plugin

Registrations

Go to https://programming-23.mooc.fi

If you're a student in University of Helsinki, use the @helsinki.fi email address. Remember to add your student number!

Registration (cont.)

Remember to register to course in SISU

Course credits can't be assigned without registration

Deadlines

There are no weekly deadlines, all material closes at the end of the year

At least 25 % of points is required from each round to attend the exam

Passing the Course

Minimum requirements:

- 25 % of all points from each round
- 50 % of exam score

Tasks get harder during the course, so it is a good idea to try to complete as much as possible from the very beginning

Passing the Course (2)

Grade includes MOOC points (50 %) and exam score (50 %)

Total points	**Grade**
90%	5
80%	4
70%	3
60%	2
50%	1
<50%	Fail

Exams

Three exams, dates to be announced in the material. Exam dates at mooc.fi

Done electronically with own computer; open between 10.00 and 22.00

You can attend as many exams as you like

Use of AI

In this course, the use of language models is:

- **Permitted** for information retrieval or explaining and summarizing topics.
- **Permitted** for identifying or explaining errors in self-produced code.
- **Prohibited** for generating program code.
- Completely **prohibited** during the exam.

Previous Course Feedback

Lots of exercises

The course exam was too easy

Why we need to use VS Code?

Why is the deadline of exercises at the exam day?

Week 2

Introduction to Programming

12.9.2024

Last Week

- Outputting data
- Inputting data
- Variables
- Reading integer type data
- Formatting output, part 1
- Simple conditional statement

Feedback:

- Do we have Moodle page → no
- Other questions about 1st week?

Programming terminology

Statement

Block

Expression

Function

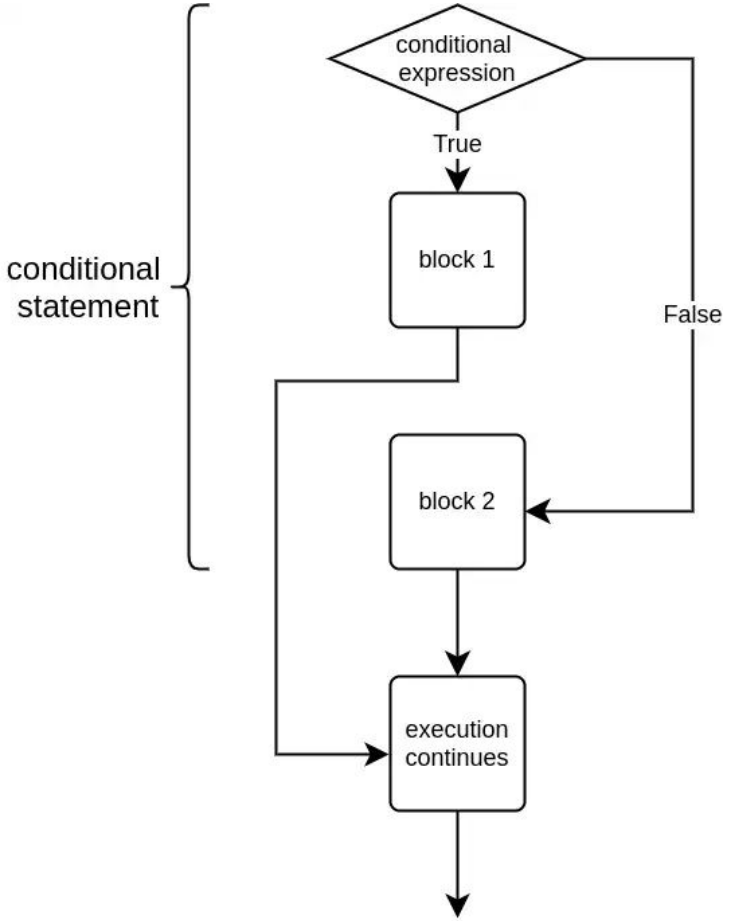
Type

Syntax (semantics)

Debugging

More conditional statements

Optional branch with else statement



Logical operators

a	**b**	**a and b**	**a or b**
False	False	False	False
True	False	False	True
False	True	False	True
True	True	True	True

a	**not a**
False	True
True	False

Chaining conditions in Python

```
a > 1 and a < 10 ⟷ 1 < a < 10
```

Nested conditions

Be careful with indentations!

```
number = int(input("Give a number: "))

if number >= 0:
    if number % 2 == 0:
        print("Number is even")
    else:
        print("Number is odd")
else:
    print("Number was negative")
```

Simple loop

Repetition with while(True) statement

Continued until **break** is encountered

Next week: more about defining conditions of repetition

Next week

More loops

String operations

Own simple functions

Week 3

Introduction to Programming

19.9.2024

Last week (week 2)

- More conditional statements
  - elif
  - else
  - logical operators
- Simple loop: while True

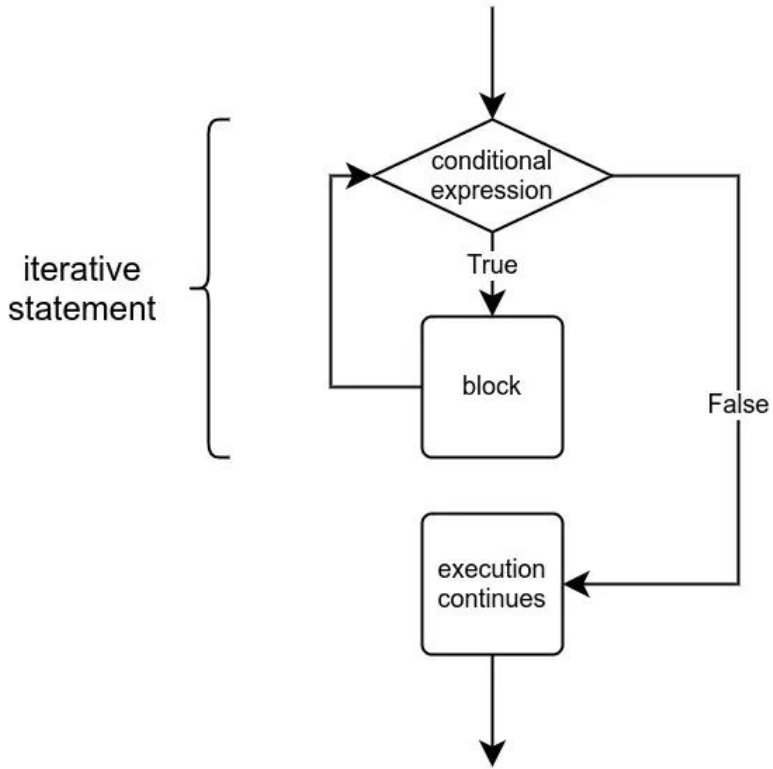
Questions

- Using external editor
- Private msgs to instructors

Repetition: while

Loop continues while the condition is True

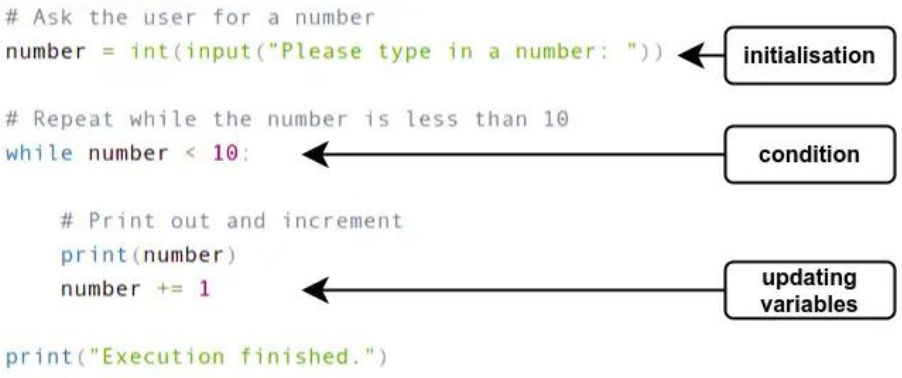
If the condition is False to begin with, the loop is not executed at all



Initialisation, Condition & Update

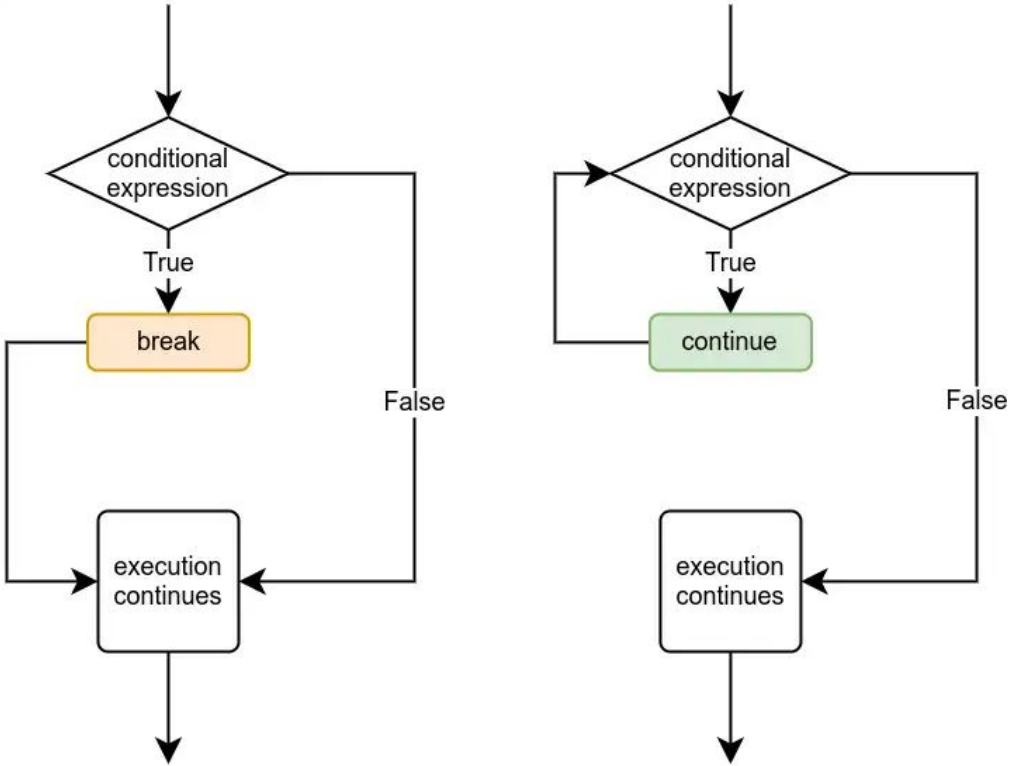
The three components of a loop

Forgetting update is a common mistake
→ may lead to "inifinite" loop




```
# Ask the user for a number
number = int(input("Please type in a number: "))

# Repeat while the number is less than 10
while number < 10:

    # Print out and increment
    print(number)
    number += 1

print("Execution finished.")
```

Break and continue



Nested loops

Pay attention to

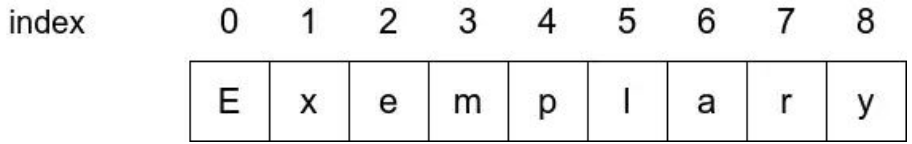
- variables and their updates
- indentation

Indexing strings

A string has characters between indices

`[0, len(string) - 1]`

```
String my_string = "Exemplary";
```

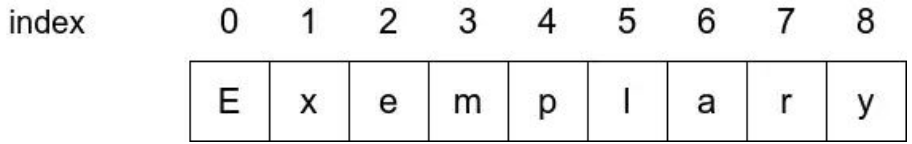


Character from a string

With brackets:

```
my_string[0] # "E"

my_string[2] # "e"

my_string[4] # "p"
```

```
String my_string = "Exemplary";
```

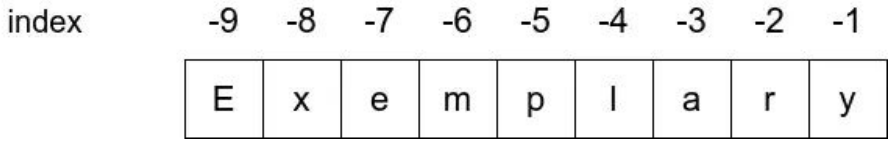


Indexing from end, negative indices

With brackets:

```
my_string[-1] # "i"

my_string[-5] # "e"

my_string[-9] # "E"
```

```
my_string = "Exemplary"
```



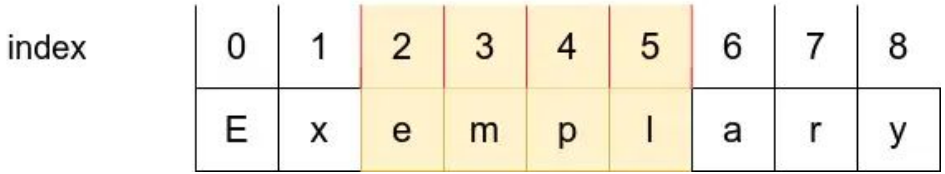
Substrings and slicing

Also with brackets

Syntax:

```
my_string[begin : end]
```

```
string = "Exemplary"
sub = string[2 : 6]
```



Finding substrings

Operator **in** returns True, if substring can be found in string, False otherwise

Method **find** returns the first index of substring in a string or -1 if the substring cannot be found

Own functions

Functions are defined as

```
def function_name():
    function code
```

Functions and parameters

Functions are defined as

```
def function_name(parameter list):
    function code
```

Testing functions

Test code inside "main program":

```
def greet():
    print("Hi!")

# Write your main function within a block like this:
if __name__ == "__main__":
    greet()
```

Next week

More functions

Lists

More tools for string manipulation

Part 4

Introduction to Programming

26.9.2024

Last week

- Repetition with while
- break and continue
- Nested loops
- Indexing strings
- Extracting characters and substrings
- Introduction to own functions

Parameters and arguments

Parameter is the variable defined in the function definition

Argument is the value passed to the function when it's called

```
def greet(name):
    print("Hello there,", name)

def sum(a, b):
    print("The sum of the arguments is", a + b)
```

```
greet("Emily")
sum(2, 3)
```

Return value

Function can return a value with **return** statement

```
number = int(input("Please type in an integer: "))
```

This way function call can be used as part of an expression

return and print

Notice the difference between **return** and **print**

```
def max1(a, b):
    if a > b:
        return a
    else:
        return b

def max2(a, b):
    if a > b:
        print(a)
    else:
        print(b)

result = max1(3, 5)
print(result)

max2(7, 2)
```

Type hints

Type	Python data type	Example
integer	`int`	`23`
floating point number	`float`	`-0.45`
string	`str`	`"Peter Python"`
Boolean value	`bool`	`True`

List

List is a collection of homogeneous items

```
my_list = []
```

New list can be created with bracket notation

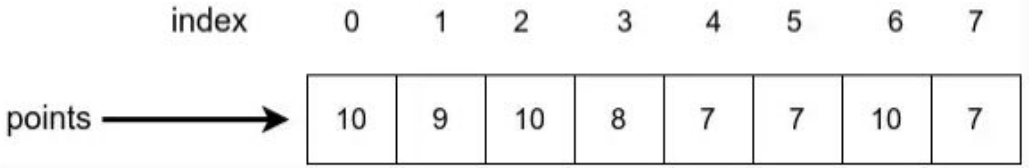
```
my_list = [7, 2, 2, 5, 2]
```

Indexing items

Items are indexed similarly to characters in a string

First item at index zero

```
points = [10, 9, 10, 8, 7, 7, 10, 7]
```



Length of a list

The length of a list can be returned with **len** function

```
my_list = [7, 2, 2, 5, 2]
print(len(my_list))
```

Again, this works similarly to strings: it returns the number of items in a list

Adding items

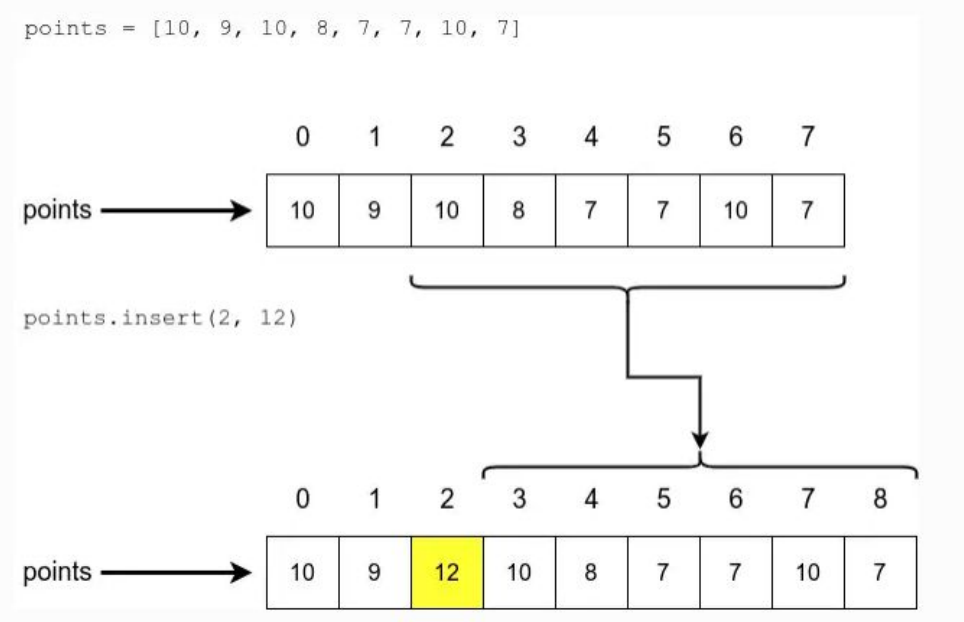
Items can be added with functions **append** and **insert**

`list.append(item)` appends at the end of the list

`list.insert(index, item)` inserts into given index

Adding items (2)

If item is inserted into middle, all following items are moved one step forward



Removing items

Method **pop** removes and returns an item at given index

Method **remove** removes item with given value - if more than one exist, only the first one is removed

Sorting lists

Method **sort** sorts the items "in place"

Function **sorted** returns a sorted copy of the list

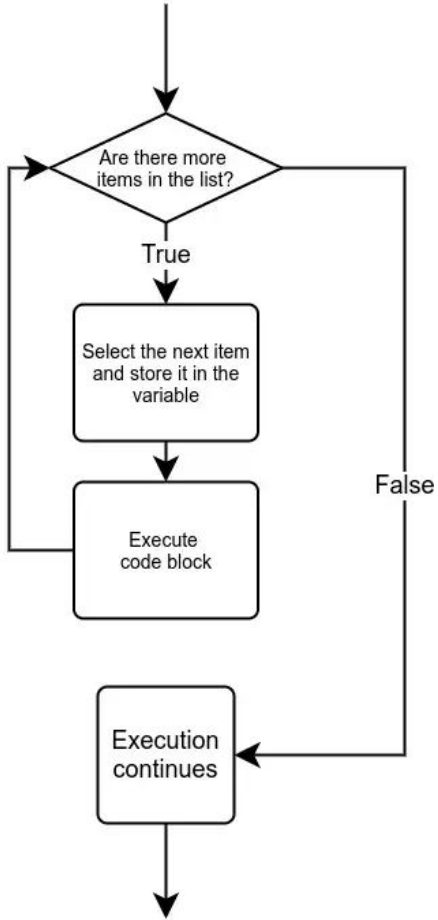
Iteration

We have used the while statement to iterate the items before

```
my_list = [3, 2, 4, 5, 2]

index = 0
while index < len(my_list):
    print(my_list[index])
    index += 1
```

Iteration with for loop

for loop is a good choice when we want to iterate through all items in a sequence


Are there more items in the list?
True
Select the next item and store it in the variable
Execute code block
False
Execution continues

Function range

Function **range** can be used to create an iterable sequence

This is handy if we want to iterate through a sequence, but do not need the list of values

```
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

Slicing lists

Syntax is similar to substrings

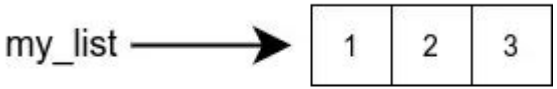
```
list[start : end]
```

or

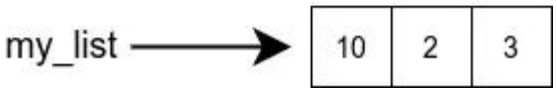
```
list[start : end : step]
```

Immutability of strings

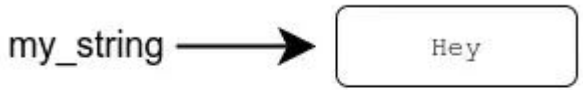
```
my_list = [1,2,3]
```



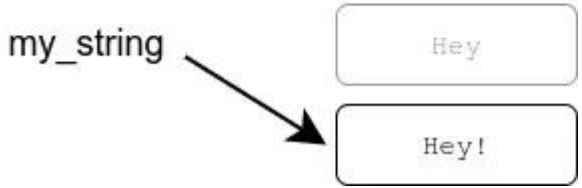

```
my_list[0] = 10
```




```
my_string = "Hey"
```




```
my_string = my_string + "!"
```



Some useful methods

Method **count** returns the number of items in a sequence

Method **replace** creates a new string with given substring replaced with another substring

Formatting output

Concatenation:

```
name = "Mark"
age = 37
print("Hi " + name + " your age is " + str(age) + " years" )
```

Separating with commas:

```
print("Hi", name, "your age is", age, "years" )
```

```
print("Hi", name, "your age is", age, "years", sep="")
```

f-strings

```
name = "Erkki"
age = 39
print(f"Hi {name} your age is {age} years")
```

```
number = 1/3
print(f"The number is {number}")
```

Next week

More lists

Multidimensional lists

List references

Dictionary

Tuple

Part 5

Introduction to Programming

3.10.2024

Last week

- parameters & arguments
- return values of functions
- type hints
- lists
- iteration with for statement
- range function
- count and replace
- f strings

About the Exercises on This Course

Becoming a proficient programmer requires a lot of practice, sometimes even quite mechanical practice. It also involves developing problem solving skills and applying intuition. This is why there are a lot of exercises of different kinds on this course. Some of them ask you to quite straightforwardly apply what you have learnt in the material, but some of them are intentionally more challenging and open-ended.

Some of the exercises might at first seem overwhelming, but this is nothing to worry about. None of the exercises is strictly mandatory, and in fact *only 25 % of the points in each part is required to pass the course.* You can find more details about passing the course on the page on grading.

The exercises are not in any specific order of difficulty. Each section usually introduces some new programming concepts, and these are then practised with both simpler and more complicated exercises. **If you come across an exercise that feels too difficult, move on to the next one.** You can always come back to the more difficult exercises if you have time later.

When the going inevitably gets tough, a word of consolation: a task that seems impossibly difficult this week will likely feel rather easy in about four weeks' time.

Items other than integers

Lists can hold any items, for example strings

```
names = ["Marlyn", "Ruth", "Paul"]
print(names)
names.append("David")
print(names)

print("Number of names on the list:", len(names))
print("Names in alphabetical order:")
names.sort()
for name in names:
  print(name)
```

Items other than integers (2)

...or floats

```
measurements = [-2.5, 1.1, 7.5, 14.6, 21.0, 19.2]

for measure in measurements:
    print(measure)

mean = sum(measurements) / len(measurements)

print("The mean is:", mean)
```

Nested lists

List items can be lists:

```
my_list = [[5, 2, 3], [4, 1], [2, 2, 5, 1]]
print(my_list)
print(my_list[1])
print(my_list[1][0])
```

Matrices

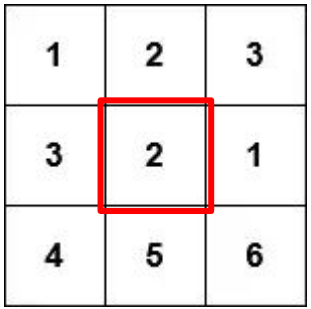
Nested lists are a good way to model matrices:

1	2	3
3	2	1
4	5	6

```
my_matrix = [[1, 2, 3], [3, 2, 1], [4, 5, 6]]
```

Referencing items

Referencing an item in the matrix now requires two indices

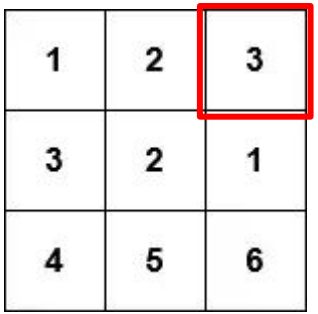


First if for **the row**, the second is for the **item in the row**

```
print(m[1][1])
```

Referencing items

Referencing an item in the matrix now requires two indices

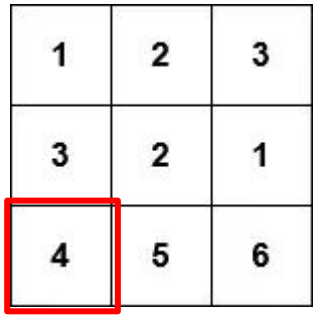


First if for **the row**, the second is for the **item in the row**

```
print(m[0][2])
```

Referencing items

Referencing an item in the matrix now requires two indices



First if for **the row**, the second is for the **item in the row**

```
print(m[2][0])
```

Iterating matrix

Iterating matrix can be done with two nested for loops

First is for iterating rows...

...and the second for iterating items in rows

```
my_matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

for row in my_matrix:
    print("a new row")
    for element in row:
        print(element)
```

Changing item values

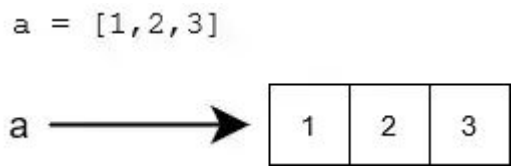
If we need to change item values, we can use two nested for loops that utilize the range function

```
m = [[1,2,3], [4,5,6], [7,8,9]]

for i in range(len(m)): # using the number of rows in the matrix
    for j in range(len(m[i])): # using the number of items on each row
        m[i][j] += 1

print(m)
```

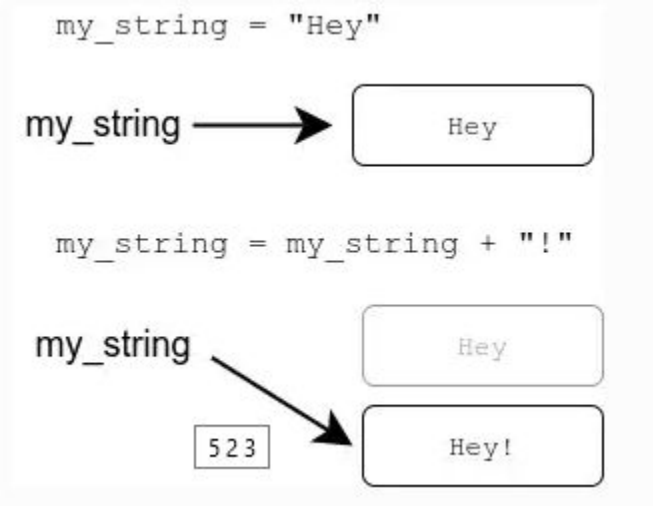
References

In Python, all variables are references

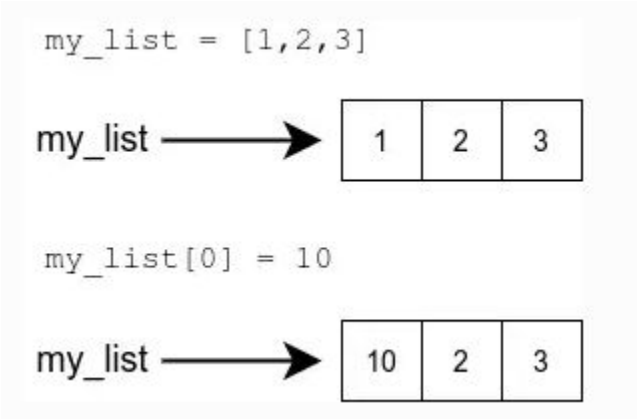
The **value** of the variable is a **reference** to an **object**



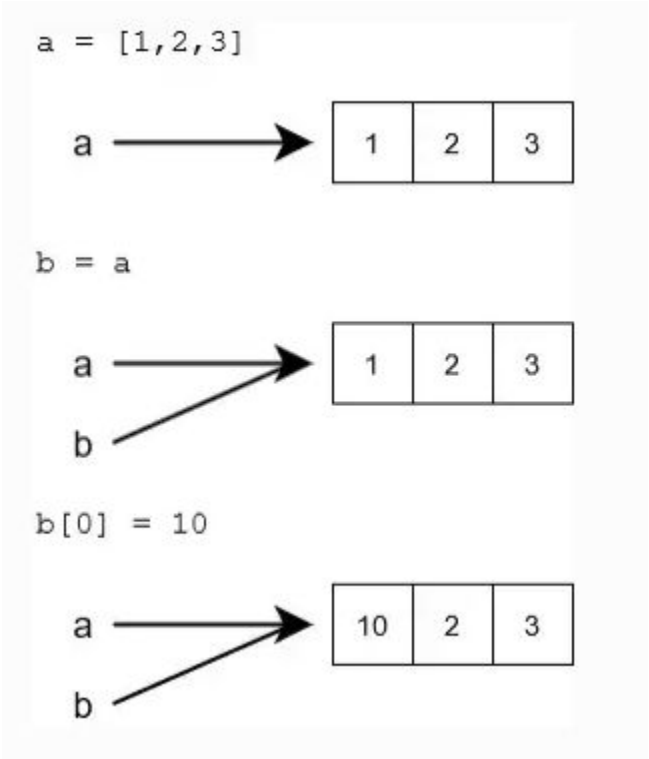
Strings are immutable



Lists are mutable



Lists are mutable (2)



Copying a list

List can be copied by taking a slice (a "sublist") which contains all items

...e.g. `my_list[0 : len(my_list)]`

...or just `my_list[:]`

List as a function argument

If the argument is a list reference, the function can change the list

```
def add_item(my_list: list):
    new_item = 10
    my_list.append(new_item)

a_list = [1,2,3]
print(a_list)
add_item(a_list)
print(a_list)
```

Side effects

If function changes a list (and this is not the main purpose of the function), this is called a **side effect**

```
def second_smallest(my_list: list) -> int:
    # in an ordered list, the second smallest item is at index 1
    my_list.sort()
    return my_list[1]

numbers = [1, 4, 2, 5, 3, 6, 4, 7]
print(second_smallest(numbers))
print(numbers)
```

Dictionary

In dictionary, all items consist of key-value pairs

Dictionary is useful (and fast) when we want to access values based on the keys

Setting items

Dictionary is initialized with curly brackets

Setting an item adds or changes the value

Items are referenced like in a list, but instead of index we use keys

```
results = {}
results["Mary"] = 4
results["Alice"] = 5
results["Kim"] = 2
```

About items

Key must be **immutable**

Hence a list for example cannot be used as a key

Iterating a dictionary

Dictionary can be iterated with a for statement

For statement iterates through the keys in the dictionary, one at a time

```
translations = {}

translations["apina"] = "monkey"
translations["banaani"] = "banana"
translations["cembalo"] =
"harpsichord"

for key in translations:
    print("key:", key)
    print("value:", translations[key])
```

Method items

If we want to iterate through both, keys and values, we can use the method **items**

```
for key, value in
translations.items():
    print("key:", key)
    print("value:", value)
```

Removing items

With **del** statement...

```
personnel = {"Antti": "principal", "Emilia": "professor", "Arto": "lecturer"}
del personnel["Arto"]
print(personnel)
```

or with a method **pop**

```
personnel = {"Antti": "principal", "Emilia": "professor", "Arto": "lecturer"}
removed = personnel.pop("Arto")
print(personnel)
print("Removed:", removed)
```

Dictionary in data grouping

Single dictionary can be used to group information that belongs together

```
person= {"name": "Paula Python", "length": 154, "weight": 61, "age:" 44}
```

Tuple

A list-like data structure

Syntactical differences:

- tuple is notated with parentheses (), list with brackets []
- tuple is immutable

Why tuple?

Tuple is a collection of values that are connected

List is a collection of similar items, and the size (or order) may change

Next week

Reading and writing files

Errors

Local and global variables

Part 6

Intro to Programming

10.10.2024

Last Week

- mixed types of items in lists
- matrices
- references
- lists as arguments and return values
- dictionary
- tuple

Reading files

Typical action in programs

Using text files is easy in Python

Other file formats may be more complicated

Example

Reading file contents

with-block handles automatic closing of file

```
with open("example.txt") as new_file:
    contents = new_file.read()
    print(contents)
```

Handling files one row at a time

Method **read** reads the contents of the entire file

Often, it's easier to read file one row at a time

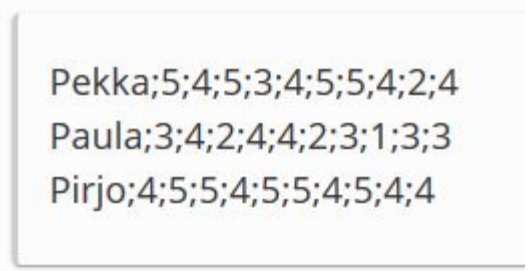
This is easy with the for statement

```
with open("example.txt") as new_file:
    count = 0
    total_length = 0

    for line in new_file:
        line = line.replace("\n", "")
        count += 1
        print("Line", count, line)
        length = len(line)
        total_length += length

print("Total length of lines:", total_length)
```

CSV Files

Comma Separated Values

Several data points in one row, separated with suitable character (i.e. a *separator*)



Splitting strings

String can be splitted into a list with the **split** method

The separator character is provided as an argument

```
text = "monkey,banana,harpsichord"
words = text.split(",")
for word in words:
    print(word)
```

Reading same file several times

File can be read in **with-**block only once

```
with open("people.csv") as new_file:
    # print out the names
    for line in new_file:
        parts = line.split(";")
        print("Name:", parts[0])

    # find the oldest
    age_of_oldest = -1
    for line in new_file:
        parts = line.split(";")
        name = parts[0]
        age = int(parts[1])
        if age > age_of_oldest:
            age_of_oldest = age
            oldest = name
    print("the oldest is", oldest)
```

Removing extra characters

Reading data from file also returns line feed characters

Easiest way to get rid of these is to use the **strip** method

```
>>> " tryout ".strip()
'tryout'
>>> "\n\ntest\n".strip()
'test'
>>>
```

Creating a file

New file can be created by providing character 'w' as the second argument for the open function

```
with open("new_file.txt", "w") as my_file:
    # code to write something to the file
```

Note, that if file exists, is will be overwritten!

Writing to file

Lines can be written to the file with the **write** method

Please note, that the linefeeds must be added manually

```
with open("new_file.txt", "w") as my_file:
    my_file.write("Hello there!\n")
    my_file.write("This is the second line\n")
    my_file.write("This is the last line\n")
```

Appending data to file

New data can be appended after the existing data in the 'a' file mode

```
with open("new_file.txt", "a") as my_file:
    my_file.write("This is the 4th line\n")
    my_file.write("And yet another line.\n")
```

Writing a CSV file

```
with open("coders.csv", "w") as my_file:
    for coder in coders:
        line = f"{coder[0]};{coder[1]};{coder[2]};{coder[3]}"
        my_file.write(line+"\n")
```

```
with open("coders.csv", "w") as my_file:
    for coder in coders:
        line = ""
        for value in coder:
            line += f"{value};"
        line = line[:-1]
        my_file.write(line+"\n")
```

Emptying and removing files

```
open('file_to_be_cleared.txt', 'w').close()
```

```
# the command to delete files is in the os module
import os

os.remove("unnecessary_file.csv")
```

Validating inputs

Typical examples of erroneous input data:

- **missing or empty input values** in mandatory fields, such as empty strings when the length of the string is critical
- **negative values** where only positive values are accepted, such as -15 as the amount of an ingredient in a recipe
- **missing files** or typos in filenames
- values that **are too small or too large**, for example when working with dates and times
- **invalid indexes,** such as trying to access index 3 in the string "hey"
- values of a **wrong type,** such as strings when integers are expected

Validating input (2)

It's often possible to prepare for erroneous data

Inputted data can be validate with conditions

```
age = int(input("Please type in your age: "))
if age >= 0 and age <= 150:
    print("That is a fine age")
else:
    print("This is not a valid age")
```

Exceptions

Some errors are difficult to check with conditional statements only

Exceptions are errors that occur during the execution of programs

Preparing for exceptions

Exceptions can be caught with try-except structure

```
try:
    age = int(input("Please type in your age: "))
except ValueError:
    age = -1

if age >= 0 and age <= 150:
    print("That is a fine age")
else:
    print("This is not a valid age")
```

Typical exceptions

ValueError

TypeError

IndexError

ZeroDivisionError

Exceptions in file handling

More than one except block

If more than one error may be thrown in the **try** block, it can be followed with more than one **except** block

```
try:
    with open("example.txt") as my_file:
        for line in my_file:
            print(line)
except FileNotFoundError:
    print("The file example.txt was not found")
except PermissionError:
    print("No permission to access the file example.txt")
```

Undefined exception

All possible exceptions can be caught by not defining the exception in the except block

Note, that this also catches the errors made by the programmer

```
try:
    with open("example.txt") as my_file:
        for line in my_file:
            print(line)
except:
    print("There was an error when reading the file.")
```

Passing exceptions

Exception thrown in a function is passed back to the caller of the function

```
def testing(x):
    print(int(x) + 1)

try:
    number = input("Please type in a number: ")
    testing(number)
except:
    print("Something went wrong")
```

Throwing exceptions

Exception can be thrown manually with keyword **raise**

```
def factorial(n):
    if n < 0:
        raise ValueError("The input was negative: " + str(n))
    k = 1
    for i in range(2, n + 1):
        k *= i
    return k
```

This is a good method for handling e.g. invalid parameters in functions

Local variables

Variables defined in functions are only visible inside functions

```
def testing():
    x = 5
    print(x)

testing()
print(x)
```

```
5
NameError: name 'x' is not defined
```

Keyword global

Variables defined in the main program can be used inside functions with keyword **global**

This is, in general, a bad idea. Instead of side effects, the functions should return a value

```
def testing():
    global x
    x = 3
    print(x)

x = 5
testing()
print(x)
```

Next week

Modules

Random

Handling times and dates

More file handling

Own modules

More about Python's features

Few words about the exam

Part 7

Introduction to Programming

17.10.2024

Last week

- Reading files
- CSV files
- Splitting strings
- Writing files
- Validating inputs
- Exceptions

course score and exams

No lecture next week, the Advanced course start on Oct 31st!

Modules

The standard library of Python contains several useful modules

We have earlier used for example module **math** to calculate square roots

Using modules

Modules are imported with the **import** statement

```
import math

# luvun 5 neliöjuuri
print(math.sqrt(5))
# luvun 8 logaritmi (2-kantainen)
print(math.log(8, 2))
```

Importing individual operations

Individual routines can be imported with **from module import operation** structure

```
from math import sqrt, log

print(sqrt(5))
print(log(5,2))
```

All routines can be imported with asterisk. This should however be avoided

```
from math import *

print(sqrt(5))
print(log(5,2))
```

Module contents

Standard library modules are described in Python documentation

```
import math

print(dir(math))
```

Operations can be listed with the function **dir**

Random numbers

Module **random** can be used to generate pseudo random numbers

```
from random import randint

for i in range(10):
    print("Dice throw: ", randint(1,6))


from random import shuffle

fruits = ["apple", "banana", "orange", "grapefruit"]
shuffle(fruits)
print(fruits)

from random import choice
fruits = ["apple", "banana", "orange", "grapefruit"]
print(choice(fruits))
```

Handling times

Whenever times and dates are handled, it is a good idea to use the standard library

Things like leap years make handling dates quite complicated otherwise

Datetime objects

Time can be modelled by creating an object from the **datetime** object

```
from datetime import datetime

aika = datetime.now()
print(aika)
```

```
from datetime import datetime

aika = datetime(1952, 12, 24)
print(aika)
```

Comparing times

Datetime objects can be compared with standard comparison operators

```
from datetime import datetime

now = datetime.now()
midsummer = datetime(2020, 6, 20)

if now < midsummer:
    print("It's not midsummer yet")
elif now == midsummer:
    print("Happy midsummer!")
else:
    print("Midsummer is gone")
```

Timedelta object

Comparing times returns a **timedelta** type object

The same object can be used to "roll" time backwards or forwards

```
from datetime import datetime, timedelta

now = datetime.now()
midsummer = datetime(2020, 6, 20)

diff = midsummer - now

print(f"It's {diff.days} days to midsummer" )

week = timedelta(days=7)
midsummer += week

print("Week after midsummer it's", midsummer)
```

Formatting times

Datetime object can be formatted with the **strftime** function

```
from datetime import datetime

aika = datetime.now()
print(aika.strftime("%d.%m.%Y"))
```

Symbol	Meaning
%d	Day (01-31)
%m	Month (01-21)
%Y	Year (YYYY)
%H	Hour (24h format)
%M	Minutes 00 to 59
%S	Seconds 00 to 59

Reading CSV files

Although CSV files are easy to handle as text, utilizing the dedicated module can be even easier

```
import csv

with open("testi.csv") as tiedosto:
    for rivi in csv.reader(tiedosto, delimiter=";"):
        print(rivi)
```

Module **csv** is great for this

Reading web pages

In Python, reading a web page (as HTML) is quite wasy

```
import urllib.request

pyynto = urllib.request.urlopen("https://helsinki.fi")
print(pyynto.read())
```

Creating own modules

Any Python program that contains functions can be used as a module

For example **words.py**

```
def first_word(sentence: str):
    parts = sentence.split(" ")
    return parts[0]

def last_word(sentence: str):
    parts = sentence.split(" ")
    return parts[-1]

def number_of_words(sentence: str):
    parts = sentence.split(" ")
    return len(parts)
```

Own module (2)

```python
def first_word(sentence: str):
    parts = sentence.split(" ")
    return parts[0]

def last_word(sentence: str):
    parts = sentence.split(" ")
    return parts[-1]

def number_of_words(sentence: str):
    parts = sentence.split(" ")
    return len(parts)
```

```python
import words

s = "Hello all it's good to meet you"
print(words.first_word(s))
print(words.last_word(s))
print(words.number_of_words(s))
```

Main program

If module contains program code outside functions, it is always executed when the module is imported

This can be prevented by including code inside `if __name__ == "__main__"` block

Some useful techniques

In Python, simple if-else conditions can be written with as a single expression

```
if x % 2 == 0:
    print("Even")
else:
    print("Odd")


print("Even" if x % 2 == 0 else "Odd")
```

Pass

Statement **pass** does not do anything

```
def testi():
    pass
```

It is still useful, as Python does not allow empty blocks

Default parameter value

Function parameters can be provided default values

In such cases, the argument may be omitted when calling the function

```
def tervehdi(nimi="Emilia"):
    print("Moikka,", nimi)

tervehdi()
tervehdi("Erkki")
tervehdi("Matti")
```

Changing number of parameters

The parameter number can be defined as changing by typing an asterisk in front of parameter name

```
def test(*values):
    print(f"You gave {len(values)} arguments" )
    print(f"The sum is {sum(values)}" )


test(1,2,3)
test(10,20,30,40,450,60)
```

Advanced course

Object oriented programming

Functional programming

Programming games

About exam

Duration 4 hours (or 5, if extra time allowed)

You can start any time between 10 and 22. Please note, that the exam ends at 22 the latest regardless of when you started.

3 to 4 coding tasks

Exercises are not too difficult

Assessed manually, small errors don't count

Example question

"Write a program which queries the user for strings, one at a time. The program then proceeds to output how many strings were given, which was the longest, how long it was and what was the most frequently used character"

…including example execution etc.

Course Introduction

Advanced Course in Programming

3.11.2023

Outline

- 5 ECTs.
- 3.11. - 16.12.2021
- One "lecture" per week at Thursdays, 10.15
  - **Exception**: no lecture next Thursday (9.11.); instead lecture on Monday, Nov. 13th at 12.15)!
- Additional workshop times, list in MOOC.fi

- Focus on independent work

Lectures

Lectures

Date	Time
Th 2/11/2023	10:15 AM–12:00 PM
Mo 13/11/2023	12:15 PM–2:00 PM
Th 16/11/2023	10:15 AM–12:00 PM
Th 23/11/2023	10:15 AM–12:00 PM
Th 30/11/2023	10:15 AM–12:00 PM
Th 7/12/2023	10:15 AM–12:00 PM
Th 14/12/2023	10:15 AM–12:00 PM

Teachers

Erkki
Timo
Senior and junior instructors

- erkki.kaila@helsinki.fi - use email primarily
- ohjelmoinnin-mooc@helsinki.fi - concerning material (GIT issues or PR's work also nicely!)
- C213, Exactum (rarely)
- +358 50 3454 325 (if I remember to switch it on occasionally)

Contents

Week	Topic
8	Objects, references, writing own classes
9	More than one class, getters and setters, visibility, static members
10	Inheritance and class hierarchies
11	List comprehension, recursion
12	Generator functions, lambda, reduce, map, filter
13 to 14	Game programming with Pygame

Initial Exam

If you know programming already and haven't completed similar course with MOOC.fi or in any other Finnish higher institute, you can pass the course with initial exam

Exam takes place at Friday 3.11. between 12 and 14 online

Register via email erkki.kaila@helsinki.fi **before tomorrow morning** 08.00! I send the instructions by email **before the exam starts.**

What Do We Do in Course?

Two components:

1. Thursday (except one Monday) shared sessions (such as this)
2. Completing exercises in MOOC.fi

Finally, an exam

Workload

Learning to program requires a lot of work

Workload is around 10 to 20 h per week

...although with previous experience the first weeks may be completed faster

Course Material

All course material is found here

https://programming-23.mooc.fi/

Workshops

Exercises are completed independently, but you don't need to complete them alone

Help can be asked in workshops and in Discord, detailed information is found in MOOC.fi

Registrations

Go to https://programming-23.mooc.fi

If you're a student in University of Helsinki, use the @helsinki.fi email address. Remember to add your student number!

If you have already registered for Introduction to Programming, no need to do it again

Registration (cont.)

Remember to register to course in SISU

Course credits can't be assigned without registration

Deadlines

All rounds are already published

There are no weekly deadlines, all material closes at **January 4th 2024**

At least 25 % of points is required from each round to attend the exam

Passing the Course

Minimum requirements:

- 25 % of all points from each round
- 50 % of exam score

Tasks get harder during the course, so it is a good idea to try to complete as much as possible from the very beginning

Passing the Course (2)

Grade includes MOOC points (50 %) and exam score (50 %)

Total points	**Grade**
90%	5
80%	4
70%	3
60%	2
50%	1
<50%	Fail

Exams

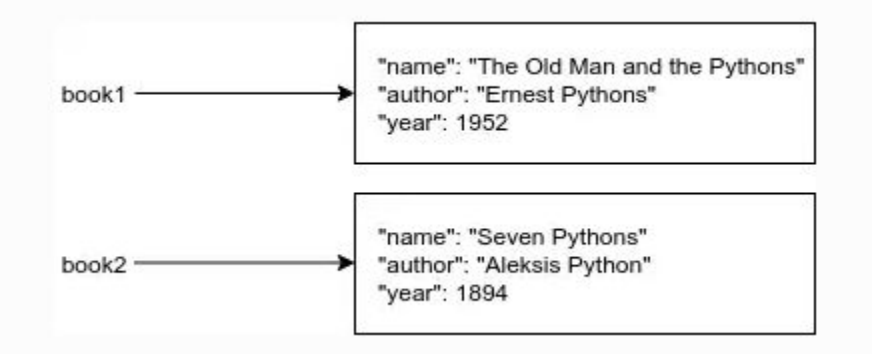
Three exams, dates are in the material. First exam at Saturday 16.12.2023

Done electronically with own computer; open between 10.00 and 22.00

You can attend as many exams as you like

Introduction to
Object-Oriented Programming

Objects are independent



Objects and methods

```
# this creates an object of type dictionary with the name book
book = {"name": "The Old Man and the Pythons", "author": "Ernest Pythons", "year": 1952}

# Print out all the values
# The method call values() is written after the name of the variable
# Remember the dot notation!
for value in book.values():
    print(value)
```

Creating objects

```
# Lists are declared with square brackets
my_list = [1,2,3]

# Strings are declared with quotation marks
my_string = "Hi there!"

# Dictionaries are declared with curly brackets
my_dict = {"one": 1, "two:": 2}

# Tuples are declared with parentheses
my_tuple = (1,2,3)
```

Constructor

```
# we are using the class Fraction from the module fractions
from fractions import Fraction

# create some new fraactions
half = Fraction(1,2)

third = Fraction(1,3)

another = Fraction(3,11)

# print these out
print(half)
print(third)
print(another)

# Fractions can be added together, for example
print(half + third)
```

Class and object

Class is the blueprint of the object

Classes define the **structure** and **functionality** of objects

It is (usually) possible to create several objects out of a single class

Defining own classes

Keyword **class:**

```
class NameOfClass:
    # class defition goes here
```

Naming classes

Usually named in *camel case* :

- Weekday
- BankAccount
- LibraryDatabase
- PythonCourseGrades

Adding a constructor

```python
class BankAccount:

    # The constructor
    def __init__(self, balance: float, owner: str):
        self.balance = balance
        self.owner = owner
```

Methods in own classes

```python
class BankAccount:

    def __init__(self, account_number: str, owner: str, balance: float, annual_interest:
        self.account_number = account_number
        self.owner = owner
        self.balance = balance
        self.annual_interest = annual_interest

    # This method adds the annual interest to the balance of the account
    def add_interest(self):
        self.balance += self.balance * self.annual_interest
```

Terminology

Client is the program code utilizing the class

Encapsulation means hiding the implementation from the client

Ensuring the **internal integrity** of the class

Outputting objects

Method __str__

The print function (for example) calls the method automatically

There's another method __repr__, which we will discuss later

Next week

More classes and objects

Visibility, better encapsulation

Properties, getters and setters

Static members

Week 10

Advanced Course in Programming

14.11.2024

Last week

Objects in data structures

Objects as parameters

Objects as attributes

Encapsulation

Private members

Static members

Need for specialization

```python
class Student:

    def __init__(self, name: str, id: str, email: str, credits: str):
        self.name = name
        self.id = id
        self.email = email
        self.credits = credits

class Teacher:

    def __init__(self, name: str, email: str, room: str, teaching_years: int):
        self.name = name
        self.email = email
        self.room = room
        self.teaching_years = teaching_years
```

Common superclass

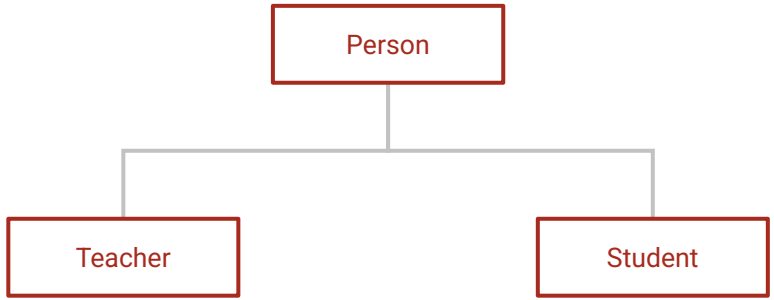
```
class Person:

    def __init__(self, name: str, email: str):
        self.name = name
        self.email = email
```

Terminology

Teacher and Student **inherit** the class Person

Person is the **superclass** of classes Teacher and Student

Teacher and Student are **subclasses** of class Person



Inheriting members

Subclass inherits all members of superclass

Subclass can refer to superclass members *if they are not private*!

```
class Bookshelf(BookContainer):

    def __init__(self):
        super().__init__()
```

Referring superclass members

Subclass can call superclass methods

```
class PlatinumCard(BonusCard):

    def __init__(self):
        super().__init__()

    def calculate_bonus(self):
        # Call the method in the base class
        bonus = super().calculate_bonus()

        # ...and add five percent to the total
        bonus = bonus * 1.05
        return bonus
```

Private members

Private members are also hidden from subclasses

```
def __init__(self):
    self._muistiinpanot = []
```

Sort of solution: protected members

Visibility

Access modifier	Example	Visible to client	Visible to derived class
Public	`self.name`	yes	yes
Protected	`self._name`	no	yes
Private	`self.__name`	no	no

Class type object

Class can return an object of it's own type

```
class Product:
    def __init__(self, name: str, price: float):
        self.__name = name
        self.__price = price

    def __str__(self):
        return f"{self.__name} (price {self.__price})"

    def product_on_sale(self):
        on_sale = Product(self.__name, self.__price * 0.75)
        return on_sale
```

Operator overloading

Overloading operators is a useful trick to enable comparing objects generated from own classes

```
class Product:
    def __init__(self, name: str, price: float):
        self.__name = name
        self.__price = price

    def __str__(self):
        return f"{self.__name} (price {self.__price})"

    @property
    def price(self):
        return self.__price

    def __gt__(self, another_product):
        return self.price > another_product.price
```

Comparison operators

Operator	Traditional meaning	Name of method
`<`	Less than	`__lt__(self, another)`
`>`	Greater than	`__gt__(self, another)`
`==`	Equal to	`__eq__(self, another)`
`!=`	Not equal to	`__ne__(self, another)`
`<=`	Less than or equal to	`__le__(self, another)`
`>=`	Greter than or equal to	`__ge__(self, another)`

Arithmetic operators

Operator	Traditional meaning	Name of method
`+`	Addition	`__add__(self, another)`
`-`	Subtraction	`__sub__(self, another)`
`*`	Multiplication	`__mul__(self, another)`
`/`	Division (floating point result)	`__truediv__(self, another)`
`//`	Division (integer result)	`__floordiv__(self, another)`

Iterators

Sometimes it would be useful to provide the functionality for iterating through own objects

Typically this could be enabled when the object contains a group of other objects

For example books in a bookshelf, students in a study registry, notes in a notebook etc.

Methods for iteration

Method__iter__ initializes the iteration

Method__next__ returns the next iterable item (if any)

```
class Book:
    def __init__(self, name: str, author: str, page_count: int):
        self.name = name
        self.author = author
        self.page_count = page_count

class Bookshelf:
    def __init__(self):
        self._books = []

    def add_book(self, book: Book):
        self._books.append(book)

    # This is the iterator initialization method
    # The iteration variable(s) should be initialized here
    def __iter__(self):
        self.n = 0
        # the method returns a reference to the object itself as
        # the iterator is implemented within the same class definition
        return self

    # This method returns the next item within the object
    # If all items have been traversed, the StopIteration event is raised
    def __next__(self):
        if self.n < len(self._books):
            # Select the current item from the list within the object
            book = self._books[self.n]
            # increase the counter (i.e. iteration variable) by one
            self.n += 1
            # return the current item
            return book
        else:
            # All books have been traversed
            raise StopIteration
```

Next week

List comprehension

Dictionary comprehension

Recursion

Week 11

Advanced Course in Programming

21.11.2024

Last week

Inheritance

About visibility

self

Operator overloading

A list out of a list

Let's assume the following problem:

We have a list where all items are strings. Each string contains digits only. We want to create a list of integers out of this.

Traditional solution

Create an empty list

Write a for statement for iterating all items one by one. Convert the items into integers with int function.

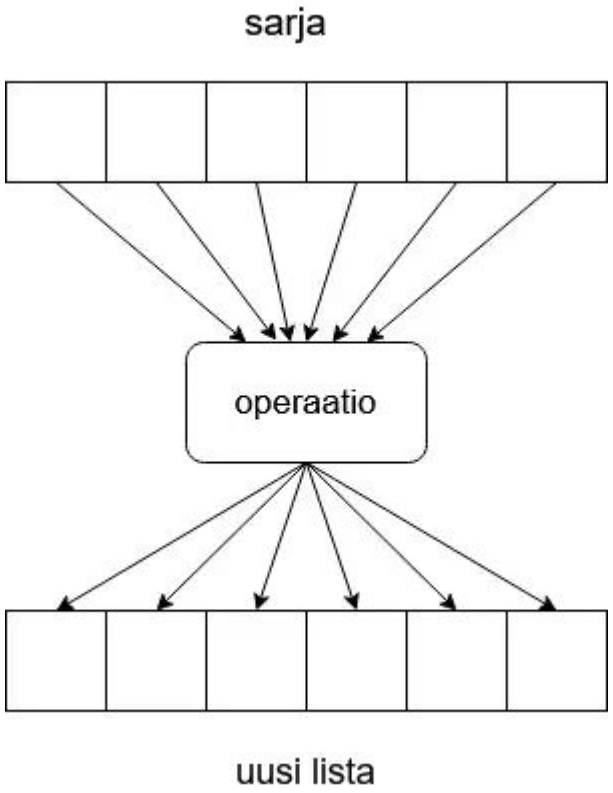
Add to new list with the append method

List comprehension

The idea is that we can create a new list with a single expression

Syntax

```
[<expression> for <item> in <sequence>]
```



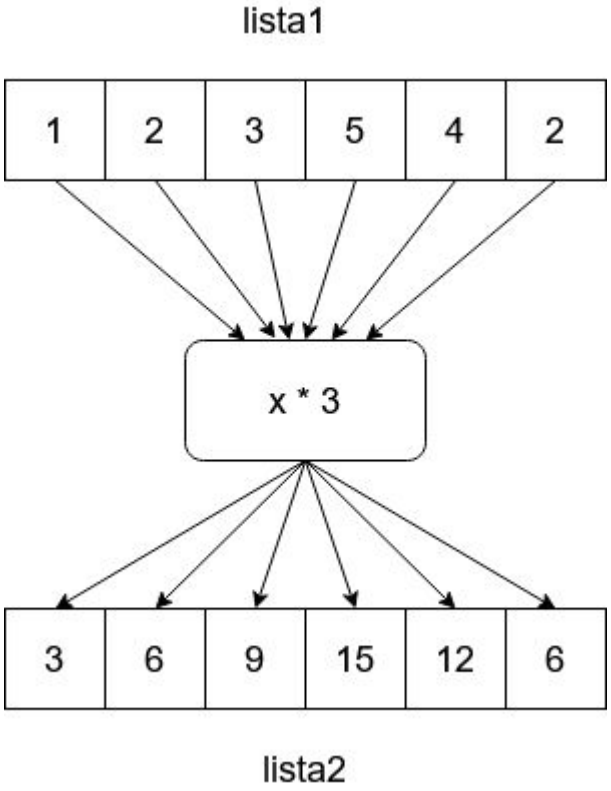
For example

Same thing as a list comprehension:

```
[int(item) for item in number_list]
```

Example

```
lista1 = [1, 2, 3, 5, 4, 2]

lista2 = [x * 3 for x in lista1]
```



Calling own function in comprehension

```
def factorial(n: int):
    """ The function calculates the factorial n! for integers above zero """
    k = 1
    while n >= 2:
        k *= n
        n -= 1
    return k

if __name__ == "__main__":
    numbers = [5, 2, 4, 3, 0]
    factorials = []
    for number in numbers:
        factorials.append(factorial(number))
    print(factorials)
```

Filtering items

General syntax:

```
[<expression> for <item> in <series> if <Boolean expression>]
```

With function

```
def factorial(n: int):
    """ The function calculates the factorial n! for integers above zero """
    k = 1
    while n >= 2:
        k *= n
        n -= 1
    return k

if __name__ == "__main__":
    numbers = [-2, 3, -1, 4, -10, 5, 1]
    factorials = [factorial(number) for number in numbers if number >= 0]
    print(factorials)
```

Alternative branch

The if expression which we discussed earlier...

```
<expression1> if <condition> else <expression2>
```

Alternative branch

...may be combined with list comprehension:

```
[<expr.1> if <condition> else <expr.2> for item in sequence]
```

Comprehension and strings

The iterable sequence
can also be e.g. a string:

```
name = "Peter Python"

uppercased = [character.upper() for character in name]
print(uppercased)
```

Comprehension and strings with join method

Useful, when the join method is combined:

```
test_string = "Hello there, this is a test!"

vowels = [character for character in test_string if character in "aeiou"]
new_string = "".join(vowels)

print(new_string)
```

Comprehension, join and split

```
sentence = "Sheila keeps on selling seashells on the seashore"

sentence_no_initials = " ".join([word[1:] for word in sentence.split()])
print(sentence_no_initials)
```

Dictionary comprehension

```
sentence = "hello there"

char_counts = {character : sentence.count(character) for character in sentence}
print(char_counts)
```

Functions can call each other

```python
def hello(name : str):
    print("Hello", name)

def hello_many_times(name : str, times : int):
    for i in range(times):
        hello(name)
```

Functions can call each other

This, however, leads to infinite loop:

```
def hello(name : str):
    print("Hello", name)
    hello(name) # function calls itself
```

Recursion

Function calls itself with changing input

```
def fill_list(numbers: list):
    """ If the length of the list is less than 10, add items to the list """
    if len(numbers) < 10:
        numbers.append(0)
        # call the function again
        fill_list(numbers)
```

Recursion and return values

Immutable values should be returned from recursive functions:

```
def factorial(n: int):
    """ The function calculates the factorial n! for n >= 0 """
    if n < 2:
        # The factorial for 0 and 1 is 1
        return 1

    # Call the function again with an argument that is one smaller
    return n * factorial(n - 1)
```

Example: Fibonacci

```
def fibo(n: int):
    if n <= 2:
        return 1

    return fibo(n - 1) + fibo(n - 2)
```

If the function is called with 1 or 2 as its argument, it returns 1, as dictated by the condition n <= 2.

If the argument is 3 or greater, the function returns the value of fibonacci(n - 1) + fibonacci(n - 2). If the argument is 3 exactly, this value is equal to fibonacci(2) + fibonacci(1), and we already know the result of both of those from the previous step. 1 + 1 equals 2, which is indeed the third number in the Fibonacci sequence.

If the argument is 4, the return value is fibonacci(3) + fibonacci(2), which we now know to be 2 + 1, which equals 3.

If the argument is 5, the return value is fibonacci(4) + fibonacci(3), which we now know to be 3 + 2, which equals 5.

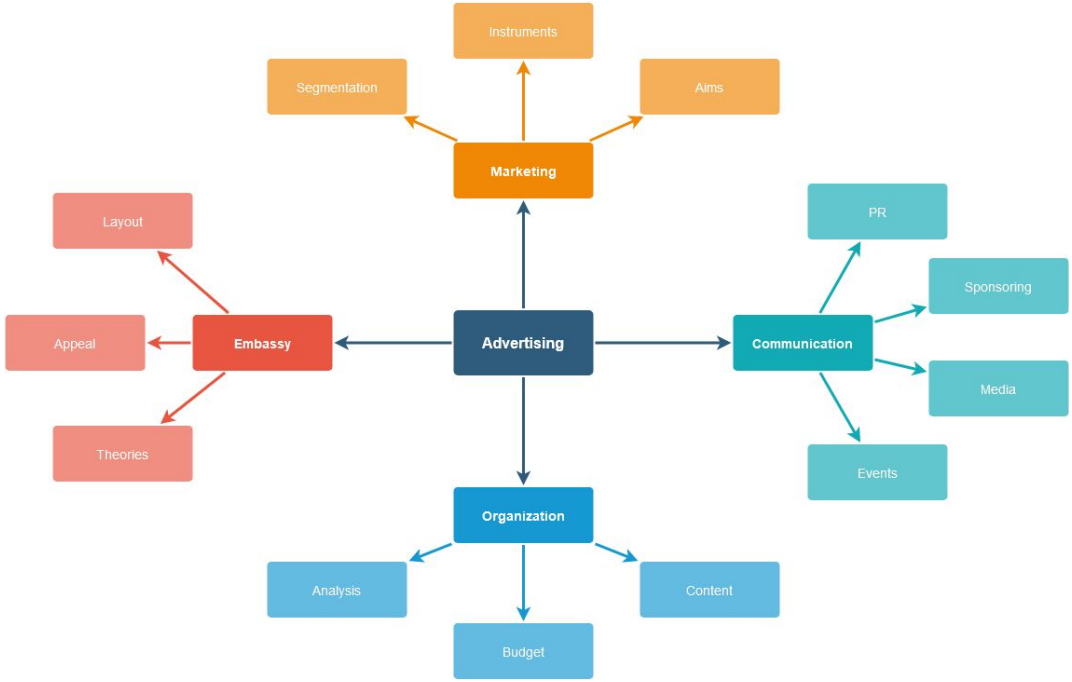
etc. etc.

Recursion: why and when?

Useful when dealing with structures which are recursive by nature

Trees, graphs, etc.

Often difficult to perceive at first



Next week

Functions as parameters

Lambda expressions

Generator functions

Functional programming: map, filter, reduce

Regular expressions

Week 12

Advanced Course in Programming

28.11.2024

Last Week

List comprehension

Filtering items in comprehensions

Dictionary comprehensions

Recursion

Sorting

Problematic if we want to sort something like tuples or our own objects

E.g. tuples are by default sorted based on their first item

Solution: provide your own value function

```
def order_by_price(item: tuple):
    # Return the price, which is the second item within the tuple
    return item[1]

if __name__ == "__main__":
    products = [("banana", 5.95), ("apple", 3.95), ("orange", 4.50), ("watermelon",

    # Use the function order_by_price for sorting
    products.sort(key=order_by_price)

    for product in products:
        print(product)
```

Defining functions inside other functions

A "helper function" that is not needed elsewhere can be defined inside other function

```
def sort_by_price(items: list):
    # helper function defined within the function
    def order_by_price(item: tuple):
        return item[1]

    return sorted(items, key=order_by_price)
```

Lambda-expression

Creates an **anonymous function**

Syntax:

```
lambda <parameters> : <expression>
```

For example

```
strings = ["Mickey", "Mack", "Marvin", "Minnie", "Merl"]

for word in sorted(strings, key=lambda word: word[-1]):
    print(word)
```

That means that…

The expression

`lambda item: item[1]`

is equivalent to the function definition

```
def price(item):
    return item[1]
```

Min and max

Functions min and max also have an optional **key** parameter

```
print("The oldest recording:")
print(min(recordings, key=lambda rec: rec.year))

print("The longest recording:")
print(max(recordings, key=lambda rec: rec.runtime))
```

Function as an argument

In Python, a function can be passed as an argument:

```
# the type hint "callable" refers to a function
def perform_operation(operation: callable):
    # Call the function which was passed as an argument
    return operation(10, 5)

def my_sum(a: int, b: int):
    return a + b

def my_product(a: int, b: int):
    return a * b
```

Generators

Sometimes it would be useful to return values from a "series" one at a time without needing to generate the entire list

For this reason, we can use *generator functions*

For example

Generator which returns values until maximum

```
def counter(max_value: int):
    number = 0
    while number <= max_value:
        yield number
        number += 1
```

StopIteration

Generator throws a StopIteration event when there are no more values to fetch

```
if __name__ == "__main__":
    numbers = counter(1)
    try:
        print(next(numbers))
        print(next(numbers))
        print(next(numbers))
    except StopIteration:
        print("ran out of numbers")
```

Generator "comprehension"

An alternative syntax for creating a generator with a single expression

```
# This generator returns squares of integers
squares = (x ** 2 for x in range(1, 64))

print(squares) # the printout of a generator object isn't too informative

for i in range(5):
    print(next(squares))
```

Functional programming

A programming paradigm where the changes in state are avoided

Lambda and expressions are examples of this

Other paradigms:

- Imperative
- Procedural
- Object-oriented

map

Performs the given operation for all items in the sequence

```
str_list = ["123","-10", "23", "98", "0", "-110"]

integers = map(lambda x : int(x), str_list)

print(integers) # this tells us the type of object we're dealing with

for number in integers:
    print(number)
```

Return value of map

map does not return a list; instead, it returns a sequence which can be iterated once

```
def capitalize(my_string: str):
    first = my_string[0]
    first = first.upper()
    return first + my_string[1:]

test_list = ["first", "second", "third", "fourth"]

# store the return value from the map function
capitalized = map(capitalize, test_list)

for word in capitalized:
  print(word)

print("print the same again:")
for word in capitalized:
  print(word)
```

filter

Only selects some of the items in the original sequence based on a condition

```
integers = [1, 2, 3, 5, 6, 4, 9, 10, 14, 15]

even_numbers = filter(lambda number: number % 2 == 0, integers)

for number in even_numbers:
    print(number)
```

reduce

Reduces the iterable sequence into a single value

```
from functools import reduce

my_list = [2, 3, 1, 5]

sum_of_numbers = reduce(lambda reduced_sum, item: reduced_sum + item, my_list, 0)

print(sum_of_numbers)
```

Regular Expressions

A "language" for filtering and searching for strings

Own syntax for defining the set of accepted strings

In Python

```
import re

words = ["Python", "Pantone", "Pontoon", "Pollute", "Pantheon"]

for word in words:
    # the string should begin with "P" and end with "on"
    if re.search("^P.*on$", word):
        print(word, "found!")
```

Rules

Alternative choices can be defined with a vertical bar

```
Please type in an expression: aa|ee|ii
Please type in a string: aardvark
Found!
Please type in a string: feelings
Found!
Please type in a string: radii
Found!
Please type in a string: smooch
Not found.
Please type in a string: continuum
Not found.
```

Rules (2)

A group of accepted characters (or substrings) is given in square brackets

```
Please type in an expression: [C-FRSO]
Please type in a string: C
Found!
Please type in a string: E
Found!
Please type in a string: G
Not found.
Please type in a string: R
Found!
Please type in a string: O
Found!
Please type in a string: T
Not found.
```

Rules (3)

Number required:

* zero or more

+ one or more

{m} exactly m

```
Please type in an expression: *1[234]5
Please type in a string: 15
Found!
Please type in a string: 125
Found!
Please type in a string: 145
Found!
Please type in a string: 12342345
Found!
Please type in a string: 126
Not found.
Please type in a string: 165
Not found.
```

Other special characters

Dot denotes any character

^ means that the match must be in the beginning

$ means that the match must be in the end

```
Please type in an expression: ^(jabba).*(hut)$
Please type in a string: jabba the hut
Found!
Please type in a string: jabba a hut
Found!
Please type in a string: jarjar the hut
Not found.
Please type in a string: jabba the smut
Not found.
```

Next Week

One more lecture.

Game programming with Pygame